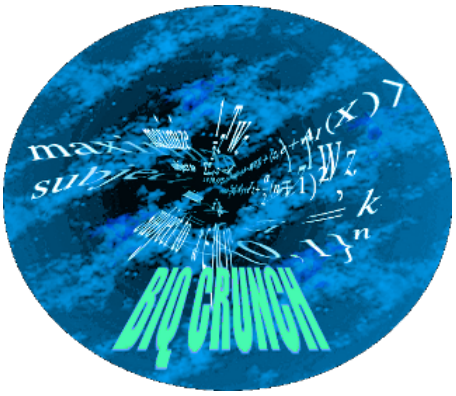
BiqCrunch User's Guide



Nathan Krislock, Jérôme Malick, Frédéric Roupin

October 29, 2014

Summary

BiqCrunch is a semidefinite-based solver for binary quadratic problems. It uses a branch-and-bound method featuring an improved semidefinite bounding procedure [8], mixed with a polyhedral approach (see [4, 5] for details).

BiqCrunch is written in `C` and `Fortran` and uses the external library for quasi-Newton bound-constrained optimization *L-BFGS-B* [2] and the branch-and-bound framework *BOB* [7]. *BiqCrunch* uses specific `BC` input files, and an LP format conversion tool is provided.

People involved with the development of BiqCrunch are

- Nathan Krislock (krislock@math.niu.edu)
- Jérôme Malick (jerome.malick@inria.fr)
- Frédéric Roupin (Frederic.Roupin@lipn.univ-paris13.fr)

Project contributors

- Marco Casazza: *BiqCrunch* web site, documentation
- Geoffrey Kozak: heuristics for the Maximum Independent Set Problem

Contents

1 BiqCrunch

BiqCrunch is released under the GNU Public License, version 3.0, as open source software available for non-commercial use. *BiqCrunch* is available at:

http://www-lipn.univ-paris13.fr/BiqCrunch/

1.1 Installation

Extract the files from the archive "`biqcrunch.tar.gz`". To compile and test *BiqCrunch* go to the `src` directory then run the following commands:

```
$ make
$ make test
```

It will produce specific binary files for each problem subdirectory located in the `problems/` directory. This allows one to execute *BiqCrunch* with a specific heuristic for each problem. If your problem does not appear, it is always possible to use the generic version of the solver in the `problems/generic/` directory, or add your own heuristic (details are given in this documentation). Installation from the source files requires either LAPACK or the Intel MKL libraries. If MKL is available then it will be used by default. When running tests (`make test`), the optimal values of the outputs are checked and printed.

1.2 Usage

To run *BiqCrunch* just use the specific problem binary version that can be found in the corresponding subdirectory in `problems/`.

```
$ ./biqcrunch [-v (0|1)] <INSTANCE> <PARAMETERS>
```

The parameter `-v` is the verbosity of *BiqCrunch* and `<INSTANCE>` is the input file in `BC` format. If flag `-v` is missing then *BiqCrunch* will use the non-verbose option. Be cautious: the verbose option can produce large output files for some problems. This option is mainly useful when testing different parameters values by giving additional information during the evaluation of each node of the search tree.

At the end of the command a parameters file is required (note that several files are provided for different problems). A complete description of these parameters is given in the "Advanced section" of this document.

1.3 Conversion tools

BiqCrunch uses a specific input file format (see the next section for a complete description). Nevertheless an LP file format conversion tool (`tools/lp2bc.py`, written in Python) is provided. Moreover, for each problem, some specific tools are also provided to convert standard instances (e.g., a graph generated by rudy) to `BC` files. To get usage information, just run the corresponding tool without parameters. All these tools are written in standard C and can be compiled in a straightforward manner.

1.4 Input format

BiqCrunch solves any problem that can be stated as

$$\begin{cases} \text{maximize} & x^T S_0 x + s_0^T x & \\ \text{subject to} & x^T S_i x + s_i^T x \leq a_i, & i \in \{1, \ldots, m_I\} \\ & x^T S_i x + s_i^T x = a_i, & i \in \{m_I + 1, \ldots, m_I + m_E\} \\ & x \in \{0,1\}^n & \end{cases} \tag{1.1}$$

where $x^T S_i x + s_i^T x$ is a quadratic function with integer coefficients, for $i = 0, \ldots, m_I + m_E$, and a is an integer vector. The problem has to be written in `BC` format which uses a sparse representation, similar to the `SDPA` format. The objective function and constraints coefficients are described as $(n+1) \times (n+1)$ matrices, linear terms being stored in the last line/column: $Q_0 = \begin{bmatrix} S_0 & \frac{s_0}{2} \\ \frac{s_0^T}{2} & 0 \end{bmatrix}$, $Q_i = \begin{bmatrix} S_i & \frac{s_i}{2} \\ \frac{s_i^T}{2} & 0 \end{bmatrix}$.

1.4.1 Example

Model

$$\begin{aligned} \text{maximize} \quad & 20x_1x_3 + 26x_1x_4 + 23x_2x_3 + 8x_2x_5 + 32x_3x_4 + 13x_4x_5 \\ \text{subject to} \quad & x_1 + x_2 + x_3 + x_4 + x_5 = 3 \\ & 12x_1x_3 + 24x_1x_4 + 14x_2x_3 + 16x_2x_5 + 28x_3x_4 + 12x_4x_5 \leq 30 \\ & x \in \{0,1\}^5 \end{aligned}$$

LP file

```
Maximize
  20 x1*x3 + 26 x1*x4 + 23 x2*x3 +
   8 x2*x5 + 32 x3*x4 + 13 x4*x5

Subject to
  x1 + x2 + x3 + x4 + x5 = 3

  12 x1*x3 + 24 x1*x4 + 14 x2*x3 +
  16 x2*x5 + 28 x3*x4 + 12 x4*x5 <= 30

Binary
  x1 x2 x3 x4 x5

End
```

BC file (generated by lp2bc.py)

```
# List of binary variables:
#    1:  x1
#    2:  x2
#    3:  x3
#    4:  x4
#    5:  x5
1 = max problem
2 = number of constraints
2 = number of blocks
6, -1
3.0 30.0
```

```
0 1 1 3 10.0
0 1 1 4 13.0
0 1 2 3 11.5
0 1 2 5 4.0
0 1 3 4 16.0
0 1 4 5 6.5
1 1 1 6 0.5
1 1 2 6 0.5
1 1 3 6 0.5
1 1 4 6 0.5
1 1 5 6 0.5
2 1 1 3 6.0
2 1 1 4 12.0
2 1 2 3 7.0
2 1 2 5 8.0
2 1 3 4 14.0
2 1 4 5 6.0
2 2 1 1 1.0
```

1.5 Output

BiqCrunch provides detailed information during the solving process and an output file is generated using the name of the input file (same directory).

Example of screen output: `./biqcrunch -v 1 example.bc biq_crunch.param`

```
Output file: example.bc.output
Input file:  example.bc
Parameter file: biq_crunch.param
Nodes = 3; Root node bound = 47.31
Maximum value = 43
Solution = { 1 2 3 }
CPU time = 0.0040 s
```

Example of output file: `./biqcrunch -v 1 example.bc biq_crunch.param`

```
*****************************************************************************
*                           BIQ CRUNCH Solver                               *
*****************************************************************************
| Copyright(C) 2010-2014 N. Krislock, J. Malick, F. Roupin                  |
| BIQ CRUNCH uses L-BFGS-B by C. Zhu, R. Byrd, J. Nocedal and BOB 1.0 by PNN |
| Team of PRiSM Laboratory.                                                 |
|                                                                           |
| L-BFGS-B is distributed under the terms of the New BSD License. See the   |
| website http://users.eecs.northwestern.edu/~nocedal/lbfgsb.html for more  |
| information. BOB is free software. For more information visit the website |
| http://www.prism.uvsq.fr/~blec/index.php?p=./rech/so&lang=fr              |
*****************************************************************************
Input file: example.bc
Solving as a MAXIMIZATION problem
Problem Size = 5 Number of equalities = 7 Number of inequalities = 1
Using Generic heuristic
BiqCrunch Parameters:
      alpha0 = 0.100000
  scaleAlpha = 0.500000
    minAlpha = 0.000050
        tol0 = 0.050000
    scaleTol = 0.900000
      minTol = 0.010000
    withCuts = 1
     gapCuts = -0.050000
        cuts = 500
     minCuts = 50
    nitermax = 2000
    minNiter = 12
    maxNiter = 100
     scaling = 1
        root = 0
      heur_1 = 1
      heur_2 = 1
      heur_3 = 1
  time_limit = 0
Heuristic 1:  Beta updated => 43
```

```
*********************************************************************************************
                                Node 1
*********************************************************************************************
Problem size 5
=============================================================================================
Iter  Time    gap     alpha     tol    nbit   Enorm    Inorm    ynorm    minCuts NCuts  NSub NAdd
=============================================================================================
  1   0.00    4.562  1.0e-01  5.0e-02   25  4.0e-02  0.0e+00  4.6e+01  -2.9e-01     0  -0   +0
  2   0.00    4.537  1.0e-01  5.0e-02    6  1.4e-02  1.7e-02  4.6e+01  -1.9e-01     0  -0   +7
  3   0.00    4.439  5.0e-02  4.5e-02   15  3.7e-02  2.3e-02  4.5e+01  -3.0e-02     7  -0   +0
  4   0.00    4.356  2.5e-02  4.1e-02   24  3.1e-02  0.0e+00  4.5e+01   5.5e-03     7  -4   +0
  5   0.00    4.333  1.3e-02  3.6e-02   15  3.1e-02  0.0e+00  4.5e+01   3.5e-02     3  -0   +0
  6   0.00    4.323  6.3e-03  3.3e-02   13  1.4e-02  0.0e+00  4.5e+01  -2.0e-03     3  -0   +0
  7   0.00    4.318  3.1e-03  3.0e-02   21  1.6e-02  0.0e+00  4.5e+01  -2.9e-02     3  -0   +0
  8   0.00    4.316  1.6e-03  2.7e-02   17  2.5e-02  0.0e+00  4.5e+01  -2.9e-03     3  -0   +0
  9   0.00    4.312  7.8e-04  2.4e-02   31  1.5e-02  1.5e-02  4.5e+01  -4.3e-02     3  -0   +0
 10   0.00    4.311  3.9e-04  2.2e-02   18  1.7e-02  0.0e+00  4.5e+01  -3.0e-02     3  -0   +0
 11   0.00    4.311  2.0e-04  1.9e-02   20  1.3e-02  1.0e-02  4.5e+01  -4.8e-02     3  -0   +0
 12   0.00    4.310  9.8e-05  1.7e-02   19  1.7e-02  0.0e+00  4.5e+01  -2.9e-02     3  -0   +0
 13   0.00    4.310  5.0e-05  1.6e-02   15  1.5e-02  0.0e+00  4.5e+01  -3.4e-02     3  -0   +0
=============================================================================================
Bound = 47.31, BFGS = 13, alpha = 5.0e-05, tol = 1.6e-02, cuts =    3, time = 0.0
fracsol[3] = 0.10
fixed : X[3] =  0

*********************************************************************************************
                                Node 2
*********************************************************************************************
Problem size 4
=============================================================================================
Iter  Time    gap     alpha     tol    nbit   Enorm    Inorm    ynorm    minCuts NCuts  NSub NAdd
=============================================================================================
  1   0.00    2.155  1.0e-01  5.0e-02   13  4.7e-02  0.0e+00  4.2e+01  -1.4e-01     0  -0   +8
  2   0.00    2.000  1.0e-01  5.0e-02    3  3.4e-02  0.0e+00  4.2e+01  -6.4e-02     8  -3   +6
  3   0.00    0.927  5.0e-02  4.5e-02   28  3.3e-01  4.9e-02  4.6e+01  -6.4e-02    11  -0   +0
Prune ! BFGS iterations = 3
=============================================================================================
Bound = 43.93, BFGS = 3, alpha = 5.0e-02, tol = 4.5e-02, cuts =   11, time = 0.0
Prune !
Depth = 1
Bound = 43 Beta = 43
fixed : X[3] =  1


*********************************************************************************************
                                Node 3
*********************************************************************************************
Problem size 4
=============================================================================================
Iter  Time    gap     alpha     tol    nbit   Enorm    Inorm    ynorm    minCuts NCuts  NSub NAdd
=============================================================================================
  1   0.00    0.901  1.0e-01  5.0e-02   24  3.2e-01  0.0e+00  6.0e+01  -6.4e-02     0  -0   +0
Prune ! BFGS iterations = 1
=============================================================================================
Bound = 43.90, BFGS = 1, alpha = 1.0e-01, tol = 5.0e-02, cuts =    0, time = 0.0
Prune !
Depth = 1
Bound = 43 Beta = 43

Nodes = 3
Maximum value = 43
Root node bound = 47.31
Solution = { 1 2 3 }
CPU time = 0.0040 s
```

2 Advanced section

2.1 BiqCrunch Parameters

In most cases, *BiqCrunch* will be efficient using default parameters. Nevertheless, to improve the performance of *BiqCrunch* for some problems it is advised to change the values of several parameters. These values (especially the bounding parameters) can actually have a huge impact on the efficiency of the solver. For more details the reader is referred to [5, 8].

General parameters

root: 1 to stop the algorithm after the evaluation of the root node (default=0). So *BiqCrunch* can be used as a simple solver to test a relaxation (computational time and gap). This option is also useful to tune other parameters by inspecting several output files for the root node (verbosity command line option `-v 1` should be used in that case);

time_limit: maximum running time in seconds. Set to 0 for no time limit (default=0). If the solver stops before solving the problem exactly then the final gap (between the worst bound in the search tree and the value of current best feasible solution found) will be provided;

heur_1: enable (1) or disable (0) the heuristic called at the beginning of the execution (default=1);

heur_2: enable (1) or disable (0) the heuristic called after each call of `L-BFGS-B` during the computation of the bound (default=1);

heur_3: enable (1) or disable (0) the heuristic called after the evaluation of a node (default=1);

Bounding parameters

alpha0: starting value of alpha (default=1e-1);

scaleAlpha: scaling value of alpha (default=0.5). This parameter controls the rate at which alpha decreases;

minAlpha: minimum value of alpha (default=5e-5);

tol0: starting value of tolerance (default=1e-1);

scaleTol: scaling value of tolerance (default=0.95);

minTol: minimum value of the tolerance (default=1e-2);

gapCuts: minimum violation value to add a cut (default=-5e-2);

withCuts: 1 to add triangle inequalities during the computation of the bound, 0 to compute the bound without the triangle inequalities (default=1);

cuts: maximum number of inequalities to add at each iteration (default=500);

minCuts: minimum number of inequalities not to decrease alpha (default=50);

nitermax: maximum number of iterations of the `L-BFGS-B` solver (default=2000);

minNiter: minimum number of `L-BFGS-B` calls (default=12);

maxNiter: maximum number of `L-BFGS-B` calls (default=100);

scaling: 1 to scale the constraints (default=1).

2.2 Instance syntax

A `BC` instance begins with some optional lines of comments, which are strings preceded by a semicolon or by a asterisk:

```
<COMMENT> ::= ; <STRING> | * <STRING>
```

The first line gives the problem type : -1 for minimization, 1 for maximization. This can be followed by other characters ignored by BiqCrunch.

```
<#MIN/MAX> ::= ; <-1> | <1> <STRING>
```

The next line defines the number of constraints, which is a positive integer such that `<INT>`$= m_I + m_E$.

```
<#CONSTRAINTS> ::= <INT> | <INT> <STRING>
```

Similar to the SDPA format, we define the number of blocks of the matrices of the input file. As seen before, this line also admits characters after the definition.

```
<#BLOCKS> ::= <INT> | <INT> <STRING>
```

In the `BC` format an instance can have 1 or 2 blocks depending on the model: if the model contains no constraints or only equality constraints, `<INT>` must be equal to 1; if the model also contains inequality constraints `<INT>` must be equal to 2.

The third entry of the instance describes the size of the blocks of the matrices.

```
<SIZE> ::= <INT_1>           | {<INT_1>}           |
           <INT_1>, -<INT_2> | {<INT_1>, -<INT_2>}
```

If the problem has no inequalities, then the size of the first block of the matrices (equal to $n+1$) is provided (`<INT_1>`). If the problem contains inequalities then the size of the second block must be given also (`<INT_2>`). It always starts with a *minus* before `<INT_2>` to indicate that the values of the blocks are only on the diagonal of the matrix, and it is equal to m_I. In the next line, the right-hand side values of the constraints are given as a sequence of values.

```
<RIGHT-HAND_SIDE> ::= <REAL_k> | <REAL_k> <RIGHT-HAND_SIDE>
```

The number of values must be equal to $m_I + m_E$ and `<REAL_k>` must be the right-hand side value of constraint k.

Finally, all the matrices that describe the objective function and the left-hand side of the constraints must be provided. The first matrix corresponds to S_0, the objective function. Each line represents to a non-zero element of the matrix.

```
<OBJ_MATRIX_EL> ::= 0 1 <INT_1> <INT_2> <REAL>
```

where the first (0) and second number (1) respectively mean that this line concerns the objective function matrix and the first block. `<INT_1>` and `<INT_2>` are the row and the column of the non-zero element of the matrix and `<REAL>` is its value. `<INT_1>` and `<INT_2>` must be greater than 0 and less or equal to $n + 1$. Then, similarly, for each constraint k the non-zero coefficients of the matrix S_k is given in sparse format:

```
<CONS_MATRIX_EL> ::= <INDEX_k> 1 <INT_1> <INT_2> <REAL>
```

where `<INDEX_k>` must be equal to k.

In the case of an inequality j, one has to provide the value of the second block of the matrix:

```
<INEQ_MATRIX_EL> ::= <INDEX_j> 2 <INT> <INT> <REAL>
```

where `<INDEX_j>` must be equal to j and `<REAL>` is either 1.0 for a $\leq$ inequality or -1.0 for a $\geq$ inequality.

2.3 Heuristics

BiqCrunch comes with specific heuristics for:

- *generic problems*;
- *k-cluster problem*;
- *max-cut problem*;
- *max-independent-set problem*.

One can also add their own heuristics. There are several folders `problems/<PROBLEM>` that refer to different optimization problems and a `problems/user` directory where it is possible to write a new heuristic. One can add new heuristics for *BiqCrunch* by simply creating more `problems/<PROBLEM>` folders. To add a heuristic for *BiqCrunch* put a new `heur.c` file inside the corresponding problem directory. An example of `heur.c` is already in the subdirectory `problems/user`.

2.3.1 Generic heuristics

BiqCrunch offers a generic heuristic which is useful when the user prefers not to write their own specific heuristic. This generic heuristic can be used for any binary quadratic problem. During the computation of the bound of the node, and after the evaluation of each node, *BiqCrunch* uses a variant of the classical randomized rounding heuristic [9, 10] that rounds to 1 the variables according to the probability provided by the fractional SDP solution. Indeed one has $0 \leq x_i \leq 1$ for any feasible solution of the SDP relaxation (see [8] for details about the relaxations used). The rounding is done by comparing each x_i to a fixed $\alpha = x_j$, for $j = 1, \ldots, n$. Then *BiqCrunch* tests if the resulting 0-1 vector is feasible

for the combinatorial problem, and updates the best current feasible solution if a better feasible solution is found. Afterwards, *BiqCrunch* generates an additional 100 random binary vectors by comparing each fractional x_i to a different random value γ, and again updates the best current feasible solution if a better feasible solution is found. At the root node we generate a random vector of values in the interval $[0, 1]$ and then we apply the variant of randomized algorithm described before. The generic heuristics are located in the `problems/generic` directory.

2.3.2 Heuristic timing

The heuristic function is called during the execution of the branch-and-bound algorithm:

1. at the beginning of the algorithm;
2. during the computation of the bound of each node of the branch-and-bound tree;
3. after the evaluation of each node of the branch-and-bound tree.

This information is saved in the function parameter `heuristic_code`, which can take three different values:

PRIMAL_HEUR: if the function is called at the beginning of the algorithm;

SDP_HEUR: if the function is called during the evaluation of the bound;

ROUNDING_HEUR: if the function is called after the evaluation of a node.

A simple use of this information is shown in the Code 2.1 taken from `heur.c`.

```
double BC_runHeuristic(Problem *P0, Problem *P, BobNode *node, int *x,
                       int heuristic_code)
{
   double heur_val = 0;
   switch (heuristic_code) {
   case PRIMAL_HEUR:
      heur_val = primal_heuristic(P0, x);
      break;
   case SDP_BOUND_HEUR:
      heur_val = sdpBoundHeuristic(P0, node, x);
      break;
   case ROUNDING_HEUR:
      heur_val = rounding_heuristic(P0, node, x);
      break;
   default:
      printf("Choosen heuristic doesn't exist\n");
      exit(1);
   }

   // return the value of the heuristic
   return heur_val;

}
```

Code 2.1: Use of the `heuristic_code` information

2.3.3 Additional functions

In addition to the heuristic function, the user must also define `BC_allocHeuristic(...)` and `BC_freeHeuristic(...)` to allocate and free the global dynamic structure. These functions are called by the solver at the beginning and at the end of the execution.

BiqCrunch also provides two useful functions for testing the solution produced with the heuristic:

- **int** `BC_isFeasibleSolution`(**int** `*sol`) which allows the user to test if the solution in the binary vector `sol` is feasible;
- **double** `BC_evaluateSolution`(**int** `*sol`) which returns the value of the objective function computed with the solution `sol`, a binary vector of size `problemSize`.

2.3.4 Data structure

In this section we report the declaration of the data structure used in *BiqCrunch* heuristics, consisting of:

- the `Problem` and `Inequality` structures that contain all the information about the problem instance (see Code 2.2);
- the `Sparse` structure that represents a matrix in sparse format (see Code 2.3);
- the `BobNode` structure which contains the information about a node of the branch and bound tree (see Code 2.5);
- the `BobSolution` structure which contains a binary solution vector (see Code 2.4).

These structures can be modified in order to include additional information for a specific problem (or to increase the maximum problem size).

```
typedef struct Problem {
   double *Q;  // Objective matrix in DENSE format
   Sparse Qs; // Objective matrix in SPARSE format
   int n; // size of Q
   Sparse *As; // list of sparse matrices for the inequality constraints
   double *a; // right-hand-side vector of inequality constraints
   int mA; // number of inequality constraints
   Sparse *Bs; // list of sparse matrices for the equality constraints
   double *b; // right-hand-side vector of equality constraints
   int mB; // number of equality constraints
   int max_problem; // 1 if it is a max problem, and 0 if it is a min problem
} Problem;

typedef struct Inequality {
   int i;
   int j;
   int k;
   int type;
   double value;
   double y;
} Inequality;
```

Code 2.2: Problem data structure

```
typedef struct Sparse {
   int *i;
   int *j;
   double *val;
   int nnz;
} Sparse;
```

Code 2.3: Data structure of a sparse matrix

```
/*
 * Maximum number of variables
 */
#define NMAX 1024

/*
 * Solution of the problem.
 * This structure defines the content of a solution of the problem.
 */
typedef struct BobSolution {
   /*
    * Vector X.
    * Binary vector that stores the solution of the branch-and-bound
    * algorithm
    */
   int X[NMAX];
} BobSolution;
```

Code 2.4: Data structure of a solution

```
typedef struct BobNode {
   /*
    * Node information.
    *
    */
   BobTNdInfo BobNdInfo; // (int Size, int Off)
   /*
    * Number of fixed variables.
    * Integer variable that stores the number of fixed variables in the
    * current node.
    */
   int level;
   int xfixed[NMAX];
   BobSolution sol;
   double fracsol[NMAX];
   BobTPri Pri; // (int Eval, int Depth)
} BobNode;
```

Code 2.5: Branch-and-bound tree node data structure

3 Examples

3.1 Max-Cut problem

Given a graph $G = (V, E)$ with edge weights w_{ij} for $ij \in E$ and $w_{ij} = 0$ for $ij \notin E$, Max-Cut is the problem of finding a bipartition of the nodes V such that the sum of the weights of the edges across the bipartition is maximized. Let $n = |V|$ be the cardinality of V; we can state Max-Cut as

$$\begin{aligned} \text{maximize} \quad & \sum_{i<j} w_{ij} \left(\frac{1 - x_i x_j}{2} \right) \\ \text{subject to} \quad & x \in \{-1, 1\}^n \end{aligned}$$

We can rewrite the problem of *Max-Cut* as

$$\begin{aligned} \text{maximize} \quad & x^T Q x \\ \text{subject to} \quad & x \in \{0, 1\}^n \end{aligned}$$

where Q is the Laplacian matrix of the weighted graph G.

3.1.1 Max-Cut heuristic

With *BiqCrunch* we provide the Goemans-Williamson random hyperplane algorithm [3]. The heuristic is run after each node evaluation to get a feasible solution.

3.1.2 Conversion tools

To simplify the creation of the `BC` instances we provide some tools to convert standard instances in sparse format to the `BC` format. All the tools can be downloaded from the BiqCrunch download page.

From Biq to BiqCrunch

To convert binary quadratic problems (e.g. [1] to a standard `BC` instance we provide the `qp2bc` conversion tool. This tool converts an instance in a standard sparse format to a valid instance for *BiqCrunch*. This tool is written in Python and can be used directly from command line:

```
$ ./qp2bc.py <BIQ_INSTANCE> > <BC_INSTANCE>
```

From Mac to BiqCrunch

To convert max-cut problems to a standard `BC` instance we provide the `mc2bc` conversion tool. This tool converts an instance in a standard sparse format to a valid instance for *BiqCrunch*. This tool is written in Python and can be used directly from command line:

```
$ ./mc2bc.py <MC_INSTANCE> > <BC_INSTANCE>
```

3.2 k-cluster problem

Given a graph $G = (V, E)$ the k-cluster problem consists of determining a subset $S \subseteq V$ of k vertices such that the sum of the weights of the edges between vertices in S is maximized.

Letting $n = |V|$ denote the number of vertices, and w_{ij} denote the edge weight for $ij \in E$ and $w_{ij} = 0$ for $ij \notin E$, the problem can be modeled as the following 0-1 quadratic problem:

$$\begin{aligned} \text{maximize} \quad & \frac{1}{2} x^T W x \\ \text{subject to} \quad & \sum_i x_i = k,\ x \in \{0,1\}^n \end{aligned}$$

where $W = (w_{ij})_{ij}$ is the weighted adjacency matrix of the graph G.

3.2.1 k-cluster heuristic

We use two types of heuristics to find a cluster with exactly k nodes. First, for the initial feasible point (before running the Branch-and-Bound), we use the classical greedy heuristic, since it gives very good feasible solutions: we remove vertices one at a time from the graph by choosing at each step the vertex with the smallest degree (or sum of the weights over the adjacent vertices). Second, during the evaluation of the bound and after running the bounding procedure on a subproblem having k' nodes added to the cluster, we add the remaining $k - k'$ nodes having the largest fractional values x_i in the SDP solution. More details can be found in [4]. Finally, to improve the solution we use a two-opt algorithm: swap two vertices (one in the k-cluster the other outside) until no progress is made.

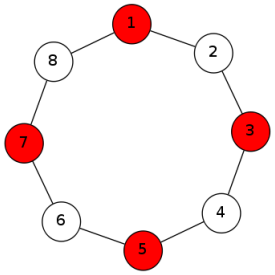
3.2.2 k-cluster instances and conversion

To obtain *BiqCrunch* input files for the k-cluster problem you can simply use the conversion tools `kc2bc` and `kcw2bc` which can convert instances from Rudy format and from the format used in [6] to `BC`. When using the conversion tool, weights can be ignored with a simple flag (thus the graph will be considered unweighted). Note that the conversion tools also add redundant constraints to the instance to improve the bound obtained during the bounding procedure [11].

3.3 Max independent set problem

Consider an undirected graph $G = (V, E)$, where $V = \{1, ..., n\}$ and $|E| = m$. To each vertex $i \in V$ a weight $w_i \in \mathbb{R}$ is assigned.

An independent set is a set $S \subseteq V$ such that no two vertices in S are joined by an edge in E. We seek an independent set of maximum total weight in G. This value is called the independent set number of G and is denoted by $\alpha_w(G)$.



$$\begin{cases} \alpha(G) = 4 \\ S_{max} = \{1,3,5,7\} \text{ or } S_{max} = \{2,4,6,8\} \end{cases}$$

Quadratic model

$$(Q) \begin{cases} \text{maximize} & \sum_{i=1}^n w_i x_i & \\ \text{subject to} & & \\ & x_i x_j = 0 & \forall ij \in E \\ & x_i \in \{0,1\} & \forall i \in V \end{cases}$$

3.3.1 Max-independent set heuristic

Two standard heuristics are provided in the corresponding `heur.c` file. First, before running the Branch-and-Bound in order to have an initial solution, sort the vertices of G by increasing degree, following this order add successively in the solution each vertex if it does not have any neighbour already in the current independent set. Second, during the evaluation of the bound and after running the bounding procedure, follow the same method as the previous heuristic by trying to add each vertex in the independent set giving a different order. This time, the order is not actually given by the increasing degree of the vertices but by the fractional solution provided by *BiqCrunch* (sort the vertices by decreasing fractional value).

Bibliography

[1] Alain Billionnet and Sourour Elloumi. Using a mixed integer quadratic programming solver for the unconstrained quadratic 0-1 problem. *Mathematical Programming*, 109:55–68, 2007. 10.1007/s10107-005-0637-9.

[2] Richard H. Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM J. Sci. Comput.*, 16(5):1190–1208, September 1995.

[3] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, 42(6):1115–1145, November 1995.

[4] Nathan Krislock, Jérôme Malick, and Frédéric Roupin. Improved semidefinite branch-and-bound algorithm for k-cluster. Available at http://hal.archives-ouvertes.fr/hal-00717212. Submitted.

[5] Nathan Krislock, Jérôme Malick, and Frédéric Roupin. Improved semidefinite bounding procedure for solving max-cut problems to optimality. *Mathematical Programming*, 143(1-2):61–86, 2014.

[6] Amélie Lambert. A library of k-cluster problems. CNAM-CEDRIC, http://cedric.cnam.fr/ lamberta/Library/k-cluster.html.

[7] Bertrand Le Cun, Catherine Roucairol, and The Pnn Team. Bob: a unified platform for implementing branch-and-bound like algorithms. Technical report, Laboratoire Prism, 1995.

[8] Jérôme Malick and Frédéric Roupin. On the bridge between combinatorial optimization and nonlinear optimization: a family of semidefinite bounds for 0-1 quadratic problems leading to quasi-newton methods. *Mathematical Programming*, 140(1):99–124, 2013.

[9] Prabhakar Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. *J. Comput. Syst. Sci.*, 37(2):130–143, October 1988.

[10] Prabhakar Raghavan and Clark D. Tompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, December 1987.

[11] Frédéric Roupin. From linear to semidefinite programming: An algorithm to obtain semidefinite relaxations for bivalent quadratic problems. *Journal of Combinatorial Optimization*, 8:469–493, 2004. 10.1007/s10878-004-4838-6.